GPU-accelerated k-mer counting

Pekka Jylhä-Ollila

Helsinki October 8, 2020

UNIVERSITY OF HELSINKI
Master's Programme in Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Study Programme in Computer Science

Tekijä — Författare — Author
Pekka Jylhä-Ollila

Työn nimi — Arbetets titel — Title

GPU-accelerated k-mer counting

Ohjaajat — Handledare — Supervisors
Simon Puglisi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	October 8, 2020	17 pages + 1 appendices

Tiivistelmä — Referat — Abstract

K-mer counting is the process of building a histogram of all substrings of length k for an input string S. The problem itself is quite simple, but counting k-mers efficiently for a very large input string is a difficult task that has been researched extensively. In recent years the performance of k-mer counting algorithms have improved significantly, and there have been efforts to use graphics processing units (GPUs) in k-mer counting. The goal for this thesis was to design, implement and benchmark a GPU accelerated k-mer counting algorithm SNCGPU. The results showed that SNCGPU compares reasonably well to the Gerbil k-mer counting algorithm on a mid-range desktop computer, but does not utilize the resources of a high-end computing platform as efficiently. The implementation of SNCGPU is available as open-source software.

ACM Computing Classification System (CCS):
Theory of computation → Design and analysis of algorithms → Parallel algorithms

Avainsanat — Nyckelord — Keywords
GPU, k-mer counting, compression

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Algorithms study track

Contents

1 Introduction

A common task in bioinformatics algorithms is k-mer counting [MK11, KDD17]. Given a string S, the problem is to count the frequency of each unique substring of length k in S. K-mer counting has several applications in bioinformatics, such as *de novo* assembly of genome sequences [BJS+02], estimating genome size and read error correction [KSS10]. Outside of bioinformatics, k-mer counting has been used to approximate string compressibility [RRRS13]. In computational linguistics, k-mers (called N-grams in that field) are also used for language models [SPHC16].

In the case of DNA sequences, the string S represents a DNA sequence, and the string alphabet is $\Sigma = \{A, C, G, T\}$. The size of the alphabet is $\sigma = |\Sigma| = 4$. Examples of 4-mers and 7-mers of the string "AGACGCTACGT" are shown in Figure 1 and Figure 2.

This thesis describes SNCGPU, a sorting-based algorithm for k-mer counting, that is designed to run on a graphics processing unit (GPU). The algorithm supports any size of k and utilizes available CPU cores. The implementation is available as open-source software at `https://gitlab.com/jype/sncgpu`.

1.1 Prior work on k-mer counting

The first algorithm that significantly improved on the k-mer counting performance of sequential algorithms was Jellyfish [MK11]. Jellyfish uses a lock-free hash table for parallel insertion and updates on a multi-core CPU. Later, algorithms like DSK [?] and KMC2 [DKGDG15] used a two-disk approach to reduce the effect of

```
AGACGCTACGT
AGAC
 GACG
  ACGC
   CGCT
    GCTA
     CTAC
      TACG
       ACGT
```

Figure 1: Example of string 4-mers.

```
AGACGCTACGT
AGACGCT
 GACGCTA
  ACGCTAC
   CGCTACG
    GCTACGT
```

Figure 2: Example of string 7-mers.

expensive I/O operations, and further improved the execution time. Currently the most efficient k-mer counter is KMC3 [KDD17], which is CPU-based. The only two existing algorithms we are aware of that use GPUs for k-mer counting are Gerbil [ERMH17] and the work by Suzuki et. al [SKIA14], with only Gerbil having a public implementation available.

2 GPU overview

Graphics processing units (GPUs) are specialized devices designed for highly parallel computing. Originally GPUs were intended only for computer graphics, but have since found use in parallel computation and machine learning. They have gained popularity in the scientific community due to their general-purpose parallel computation capabilities [OHL+08]. GPUs are typically more efficient than central processing units (CPUs) for computational tasks that process large blocks of data in parallel. Algorithms that process large amounts of data can often be adapted for GPU acceleration, although to date there has been relatively little work and basic algorithmic and data structural tools for GPUs are still missing. Recently there has been work on for example B-Tree [AAJ+19], and dynamic hash table [AFCO18] GPU data structures.

2.1 CUDA

CUDA is a high-level extension of C/C++ for general purpose computation on NVIDIA GPUs. GPUs use hundreds of parallel processor cores for executing CUDA kernels in parallel. The CUDA kernels are executed in threads, which each have their own private local memory [ND10].

Threads are organized into a grid of thread blocks, where threads can use fast shared per-block memory. Threads from different blocks in the same grid can access global GPU memory via atomic operations. The CUDA architecture is illustrated in Figure 3.

CUDA kernels are implemented in standard C language with some CUDA specific additions, such as the `__global__` modifier and built-in thread index variables. The `__global__` modifier is used to define a function that is called from the host, and executed on the GPU. Another important CUDA intrinsic function is `__syncthreads`, which specifies a synchronization point for all threads in a block. The function is

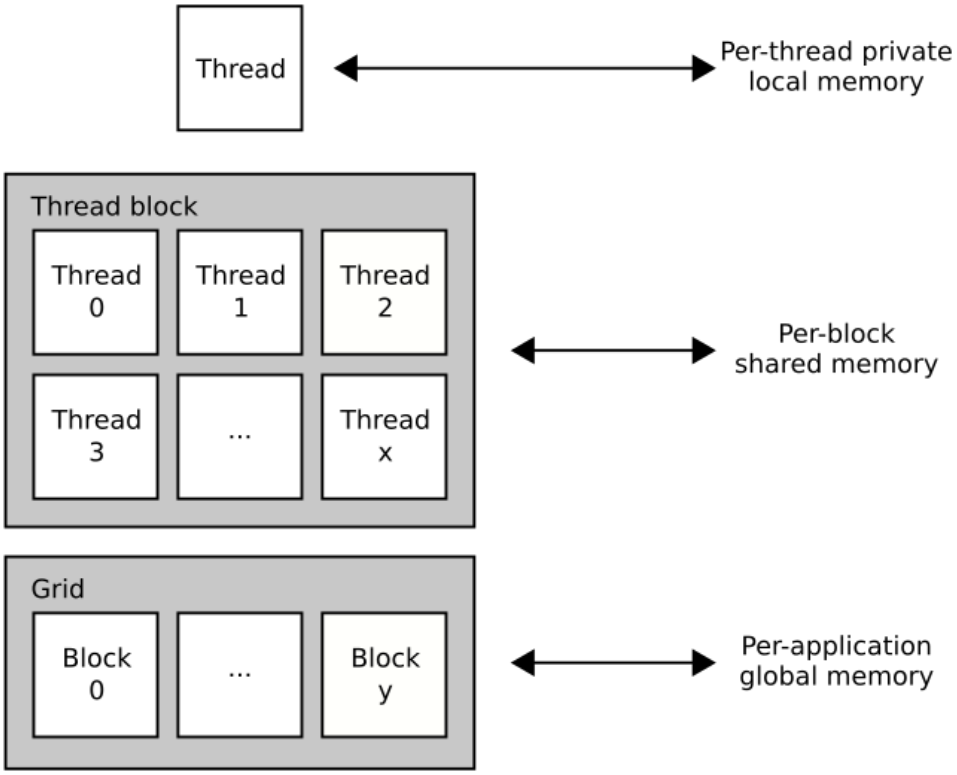


Figure 3: CUDA architecture

useful when an algorithm takes advantage of per-block shared memory, for example calculating sub-matrices for efficient matrix multiplication.

A simple SAXPY (single-precision A times X plus Y) kernel is shown in Algorithm 1. The `__global__` modifier indicates that `saxpy_kernel` is a kernel entry-point, and the function call `saxpy_kernel«<B, T»>` launches the kernel in parallel with B blocks running T threads.

Algorithm 1 CUDA SAXPY kernel

```
__global__
void saxpy_kernel(int n, float a, float* x, float* y)
{
  const int i = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < n)
    y[i] = a * x[i] + y[i];
}

void saxpy(int n, float a, float* x, float* y)
{
  // set launch configuration parameters
  int block_size = 256;
  int grid_size = (n + block_size - 1) / block_size;

  // launch saxpy kernel
  saxpy_kernel<<<grid_size, block_size>>>(n, a, x, y);
}
```

2.2 libthrust

libthrust is a parallel C++ template library for CUDA that provides an abstract interface to fundamental parallel algorithms [BH12]. The library speeds up the prototyping and development of parallel applications, while still making it possible to write specialized parts of the application with CUDA. libthrust automatically handles limits imposed by CUDA, and automates the launch configuration for maximal GPU occupancy during run-time.

User-defined operations in libthrust are written as C++ function objects. The function objects can be used to adapt generic algorithms provided by libthrust to implement user-defined operations. The example Algorithm 2 shows a SAXPY implementation using libthrust.

The libthrust `vector` generic container has an interface similar to that of the C++ Standard Template Library (STL). An STL `vector` can be uploaded to GPU memory by passing it to the libthrust `vector` constructor, or by copying its contents to GPU

memory using the thrust::copy function. The same can be done vice-versa to retrieve results from GPU memory.

Algorithm 2 libthrust SAXPY example

```
struct saxpy_functor
{
  const float a;

  saxpy_functor(float a) : a(a) {}

  __host__ __device__
  float operator()(float x, float y)
  {
    return a * x + y;
  }
};

void saxpy(float a,
     __device__ vector<float>& x,
     __device__ vector<float>& y)
{
  // setup functor
  saxpy_functor func(a);

  // call transform
  transform(x.begin(), x.end(),
      y.begin(), y.begin(),
      func);
}
```

3 Algorithm

In this section we describe the sort-and-compress GPU algorithm (SNCGPU). The sort-and-compress part is written using libthrust and can be executed on both the CPU and GPU. The algorithm uses a similar two-disk approach as some other k-mer counting algorithms. The first disk is called the *working disk*, and is used for storing temporary files. The second disk is the *input/output-disk*. It contains input data and is used to store the final output data.

3.1 Overview

The algorithm is executed in two phases. In the first phase, k-mers are parsed from the input data into bins. The size of a bin is specified by a GPU work size parameter, that should be smaller than half of the available GPU memory. The k-mers are assigned to bins by their most significant bits, so that the same k-mers

```
>SRR2003395.13958 HWI-ST365:221:C0AMHACXX:8:1101:13349:3773 length=50
AATAAACCATCTATTCAAGCTAGATCGGAAGAGCACACGTCTGAACTCCA
>SRR2003395.13959 HWI-ST365:221:C0AMHACXX:8:1101:13317:3774 length=50
TTCCTTACCTACTCTACTACAGATCGGAAGAGCACACGTCTGAACTCCAG
>SRR2003395.13960 HWI-ST365:221:C0AMHACXX:8:1101:13416:3780 length=50
CTCACTGAAGCTGGAGCTGTAGATCGGAAGAGCACACGTCTGAACTCCAG
>SRR2003395.13961 HWI-ST365:221:C0AMHACXX:8:1101:13426:3794 length=50
GGTAGACCATTAACACGTAAAAGATCGGAAGAGCACACGTCTGAACTCCA
```

Figure 4: Genome read data in FASTA format

are placed into the same bin. Since the k-mers in each bin are independent, the bins can be processed in parallel. When a bin has become full, the k-mers in the bin are sorted and run-length encoded (RLE), so that each k-mer appears only once, followed by the number of times it occured in the bin. The sorted RLE k-mers are then written to a temporary file. A single bin may produce multiple temporary files.

In the second phase, the k-mers for each bin are read from temporary files. The sequences in each bin are merged using a multiway merging algorithm, and again sorted and run-length encoded. The resulting sequence is written to a final output file. At the end of the second phase, the output for each bin contains the sorted k-mers and their corresponding frequencies.

3.2 Input processing

The pipeline starts with parsing FASTA format input into k-mers. An example of FASTA format data is shown in Figure 4. K-mers are encoded as one or more 64-bit integers by the input parser. The number of integers required to encode a k-mer is $\lceil k\lceil \log_2 \sigma \rceil / 64 \rceil$. The remaining unused bits are set to 0 to maintain sorting order. For DNA sequences, the parser encodes the alphabet $\Sigma = \{A, C, G, T\}$ as binary 00, 01, 10 and 11 respectively.

For a single k-mer, the integers that are used to represent the k-mer are stored in arrays called channels. The least significant integer is stored in the first channel, and the most significant integer in the last channel. This improves sorting performance with large amounts of k-mers by representing them as a Structure of Arrays [BH12].

The channels form a sequence of k-mers, which is stored in a bin. A bin is a sequence where all k-mers have the same most significant bits; this improves the compression ratio when similar k-mers are placed in the same bin. The data structures are illustrated in Figure 5. The example shows four bins, but the number of bins can be adjusted to be any power of two.

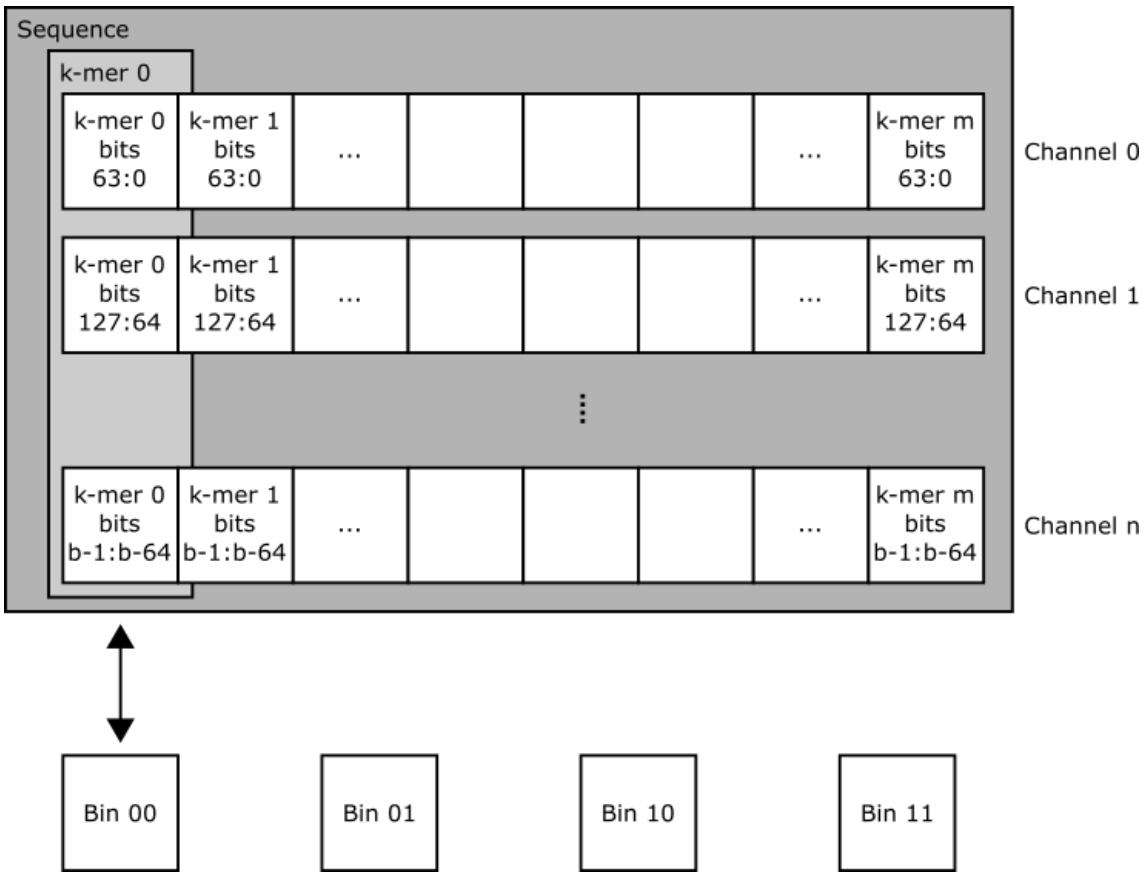


Figure 5: Bins, sequences, channels and k-mers.

Input processing is done in a single thread that parses k-mers into bins from the FASTA input data. The bin is full when its size reaches the specified GPU work size. Once a bin is full or the input has been consumed, the bin is placed into the sort and compress queue for further processing.

3.3 Sort and compress

The sort and compress step sorts a sequence of k-mers and applies run-length encoding. Using libthrust, the sorting and compression can be carried out using both the CPU and GPU.

The sequence is then sorted using a lexicographical sorting algorithm described below. The algorithm creates a permutation vector, which will contain the indices of each k-mer in the sorted sequence. The permutation vector is first initialized to the sequence $0..n$. For each channel in the sequence, the vector is updated by doing a stable sort on the permutation vector, with the channel values as the sort keys. The final permutation vector is applied to each channel to get the sorted sequence. The lexicographical sorting algorithm is shown in Algorithm 3.

After sorting the k-mer sequences, the sequences are run-length encoded. Run-length encoding is a data compression algorithm, which stores consecutive repeated values as a single value along with the number of occurences. The algorithm is trivial to implement on the CPU, but not as straight-forward on the highly parallel GPU.

The paper [Bal] describes how run-length encoding can be implemented as a parallel algorithm. In short, the algorithm takes an integer array `in` as input and produces an array `flags`, where a 1 means the adjacent elements in `in` were equal and 0 if non-equal. A prefix sum is calculated for the `flags` array to produce `rle_indexes`, which contains indexes to `in` from where elements will be copied. Finally, the run-length encoded output is produced by doing a `scatter` operation with `in` as the input and `rle_indexes` as the mapping. In the `scatter` operation, the run-length for each symbol can be calculated from `rle_indexes` by subtracting adjacent elements.

The algorithm shown here is an extended version, as the encoding has to be applied to multiple channels in the sequence, and their lengths. The run-length encoding algorithm is shown in Algorithm 4 as pseudocode.

After run-length encoding has been applied to the sorted sequence, the sequence contains only unique k-mers. The RLE k-mer sequences are copied back to host

Algorithm 3 Parallel lexicographical sort

```
// Based on libthrust example lexicographical_sort.cu
template<typename KeyVector, typename PermutationVector>
void update_permutation(KeyVector& keys, PermutationVector& permutation)
{
  // Temporary storage for keys
  KeyVector tmp(keys.size());

  // Permute the keys with the current reordering
  thrust::gather(permutation.begin(), permutation.end(),
            keys.begin(), tmp.begin());

  // Stable sort the permuted keys and update the permutation
  thrust::stable_sort_by_key(tmp.begin(), tmp.end(),
            permutation.begin());
}

template<typename KeyVector, typename PermutationVector>
void apply_permutation(KeyVector& keys, PermutationVector& permutation)
{
  // Copy keys to temporary vector
  KeyVector tmp(keys.begin(), keys.end());

  // Permute the keys
  thrust::gather(permutation.begin(), permutation.end(),
            tmp.begin(), keys.begin());
}

template<typename KeyVector, typename ValueVector>
void sort_sequence(std::vector<KeyVector>& seq,
       ValueVector& lengths, uint64_t n)
{
  if (seq.size() == 1)
  {
    // Regular sort_by_key is sufficient
    thrust::sort_by_key(seq[0].begin(), seq[0].end(),
             lengths.begin());
    return;
  }

  KeyVector permutation(n);
  thrust::sequence(permutation.begin(), permutation.end());

  // Get permutation vector for sequence
  for (uint32_t i = 0; i < seq.size(); ++i)
    update_permutation(seq[i], permutation);

  // Apply permutation to each channel
  for (uint32_t i = 0; i < seq.size(); ++i)
    apply_permutation(seq[i], permutation);

  // Apply permutation to run lengths
  apply_permutation(lengths, permutation);
}
```

Algorithm 4 Parallel run-length encoding

```
function AdjacentDifference(channel[n])
    diff ← vector(n)
    diff[0] ← 1
    for i ← 1 to n do
        diff[i] ← channel[i] ≠ channel[i-1]
    end for
    return diff
end function
function ScanSequence(sequence[m][n])
    combined ← AdjacentDifference(sequence[0])
    for i ← 1 to m do
        diff ← AdjacentDifference(sequence[i])
        combined ← combined ∨ diff
    end for
    return combined
end function
function RunLengthEncode(sequence[m][n],lengths[n])
    encoded ← vector(m,n)
    encodedLengths ← vector(n)
    combined ← ScanSequence(sequence)
    indices ← combined · 0..n
    remove indices[i] where indices[i] = 0
    for i ← 0 to m do
        encoded[i] ← gather indices from sequence[i]
    end for
    combined ← inclusive scan of combined
    encodedLengths ← reduce lengths by key combined
    return encoded, encodedLengths
end function
```

Table 1: simple-8b selector options.

Selector value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ItemWidth	0	0	1	2	3	4	5	6	7	8	10	12	15	20	30	60
GroupSize	240	120	60	30	20	15	12	10	8	7	6	5	4	3	2	1
Wasted bits	60	60	0	0	0	0	0	0	4	4	0	0	0	0	0	0

memory, and placed into the output queue.

3.4 Output encoding

A common bottleneck in k-mer counting is storage I/O performance. Reading and writing several gigabytes of data is usually more expensive than the actual counting. The amount of storage I/O can be reduced by applying an encoding and decoding to the data that is written to storage. The sort-and-compress GPU algorithm does post-processing and variable-length encoding on k-mers, and uses Simple-8b encoding for k-mer frequencies.

As a post-processing step, we take the adjacent difference of each k-mer in the sequence to improve the compression ratio. This is done as an arbitrary precision integer substraction between each k-mer in the sequence.

The integers that represent each k-mer are encoded with variable length quantity (VLQ) encoding [SWYZ02]. VLQ encoding uses a variable number of bytes to represent arbitrarily large unsigned integers. In each byte, 7 bits are used to store integer data, and the most significant bit marks the continuation of bytes. If the most significant bit in the current byte is 0, then the byte is the last byte of the VLQ encoded integer. If the bit is 1, then another byte follows. For example, the integer 1234 (100 11010010 in binary) would be encoded as **1**0001001 **0**1010010.

K-mer frequencies are encoded with Simple-8b encoding [AM10]. Simple-8b is a fast integer encoding for packing variable quantities of integers into 64-bit words. The encoding uses a 4-bit selector that specifies the common length of integers in each 64-bit word. The selector options are shown in Table 1. In the figure, `GroupSize` is the number of integers encoded to the 64-bit word. `ItemWidth` is the number of bits used per encoded integer. For example, a selector value of 4 would mean that there are 20 3-bit integers encoded into the 64-bit word.

After encoding k-mers with Simple-8b encoding and frequencies with VLQ encoding,

Table 2: Data sets.

Data set	No. of bases (G)	Size (GB)	Avg. read length
M. Balbisiana	56.3	98.6	100.0
H. Sapiens	121.4	166.0	151.0
GRCh38	3.0	3.3	1000.0
Random	10.0	10.2	1000.0

the k-mers and frequencies are written to per-bin temporary files.

3.5 Multiway merge

At the end of the first phase, the sorted k-mers and their corresponding frequencies have been written to the working disk. The second phase does a multiway merge on the sequences in each bin, and creates the final output to the input/output-disk.

The second phase re-uses the sort and compress algorithm from Section 3.3. The main difference is that input is read from multiple temporary files, and merged using a multiway merging algorithm into one sequence per bin. The final output is also one sequence per bin instead of multiple temporary files.

The multiway merge reads sorted k-mer sequences and frequencies from temporary files up to the specified GPU work size. All k-mers less than or equal to the smallest k-mer in the sequences are inserted into a merged sequence along with their frequencies. The merged sequence is placed into the sort and compress queue. This is repeated until all input has been processed.

In the second phase, sequences are sorted and compressed as in the first phase. The final output is sorted pairs of encoded k-mers and corresponding frequencies per bin.

4 Results

The performance of SNCGPU was compared against Gerbil with data sets shown in Table 2. The random data set consists of 10GB uniformly random reads of length 1000. The data sets section in the appendix contains links to the other data sets that were used.

Tests were run on two test systems; system one is a mid-range desktop computer,

Table 3: Test systems.

	System One	System Two
CPU	Intel Core-i5 2500K (4 cores)	Intel Xeon (32 cores)
RAM	8 GB DDR3	360 GB DDR3
GPU	Nvidia GeForce GTX 660 (2 GB)	Nvidia Tesla V100 (16 GB)
Working disk	1 TB SATA SSD	1.5 TB NVMe SSD
I/O disk	480 GB SATA SSD	1.5 TB NVMe SSD

and system two is a high-performance computer in the Finnish Grid and Cloud Infrastructure (persistent identifier urn:nbn:fi:research-infras-2016072533). Table 3 shows details on the test system hardware configurations.

Test results are shown in Table 4. The only time SNCGPU was slightly faster was with $k = 28$ with the data sets GRCh38 and random data on test system one.

The M. Balbisiana data set has a shorter read length, which means that for large k the running time decreases as there are less k-mers to process. The running time increases more for both SNCGPU and Gerbil when k crosses multiples of 32 due to changes in the internal representation of k-mers.

The test results show that the performance of SNCGPU is comparable to Gerbil on test system one with small data sets, but on test system two SNCGPU does not utilize the available resources as efficiently as Gerbil. This is likely due to the fact that sorting is a more memory intensive operation than hashing, which Gerbil uses. System two has much more memory available, but the bandwith and latency are roughly the same as on system one, which becomes a bottleneck.

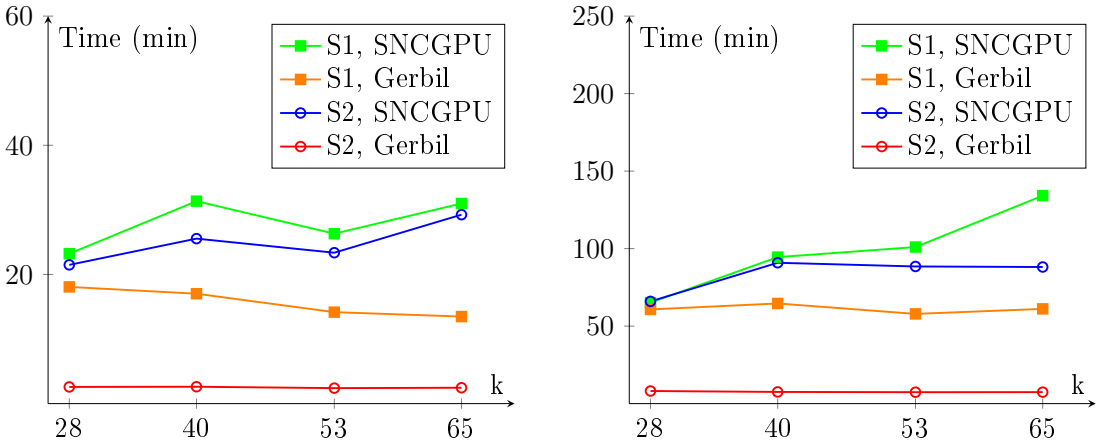
Figure 6 shows running times for the M. Balbisiana and H. Sapiens data sets on both test systems. It can be seen that SNCGPU does not gain much at all from the available hardware resources on test system two, while Gerbil is on average seven times faster than it is on test system one.

The optimal number of k-mer counting CPU threads for SNCGPU on both test systems was determined to be four, where one thread uses the GPU for counting. Using any more CPU threads decreased the running time. The reason is likely because of cache trashing caused by the sorting algorithm memory usage patterns. The hashing algorithm used by Gerbil has much more linear memory usage patterns which scale better with the greater number of CPU cores.

Storage device read and write speed have a large effect on the running time of

Table 4: Running times in seconds.

		System one		System two	
Data set	k	SNCGPU	Gerbil	SNCGPU	Gerbil
M. Balbisiana	28	1392.60	**1084.24**	1288.97	**154.92**
	40	1880.57	**1020.96**	1532.73	**156.74**
	53	1578.70	**849.66**	1401.92	**143.84**
	65	1859.30	**808.65**	1755.27	**147.39**
H. Sapiens	28	3909.32	**3645.42**	3957.48	**487.90**
	40	5670.06	**3876.77**	5451.88	**452.29**
	53	6060.71	**3472.72**	5308.86	**442.29**
	65	8048.00	**3667.11**	5287.78	**444.43**
GRCh38	28	**127.50**	146.22	111.35	**30.79**
	40	200.46	**162.50**	129.69	**31.96**
	53	238.45	**186.83**	143.58	**37.60**
	65	331.39	**231.90**	211.13	**45.17**
Random	28	**424.26**	594.17	276.08	**69.91**
	40	654.47	**631.75**	368.56	**73.51**
	53	819.26	**672.64**	383.73	**92.10**
	65	1081.56	**730.90**	557.35	**107.16**



(a) Results for M. Balbisiana data set.

(b) Results for H. Sapiens data set.

Figure 6: Running times on test systems one (S1) and two (S2).

Gerbil. The encoding used by Gerbil is simple and fast but has a low compression ratio. Gerbil could benefit from a better choice of encoding to increase the storage throughput. SNCGPU reads and writes less data due to its output encoding so its performance does not depend as much on the storage device speed.

The optimal encoding depends largely on the CPU and storage speed. The VLQ and Simple-8b encodings were determined to be the best for SNCGPU after testing on test system one, but different encodings might have worked better on test system two.

During the first phase, SNCGPU spends most of its time waiting for input bins to fill up instead of actual counting or writing output. Not much time was spent on optimizing the input processing code, which could have improved the performance significantly.

5 Conclusions

The GPU-accelerated SCNGPU k-mer counting algorithm was presented. Testing on two systems showed that the performance of SNCGPU on a mid-range desktop computer is moderate, but it does not perform as well as Gerbil on high-end computing platforms.

Gerbil could benefit by utilizing a similar compression scheme as SNCGPU. This would reduce the effect of the storage I/O performance bottleneck that limits the performance of Gerbil.

As future work, SNCGPU could be improved to utilize multiple GPUs, and a CPU sorting algorithm that scales better with the number of available CPU cores. Using a different k-mer counting algorithm on the CPU than GPU could also be a better solution.

6 Acknowledgements

The test results were produced using computational resources from the Finnish Grid and Cloud Infrastructure (persistent identifier urn:nbn:fi:research-infras-2016072533).

References

AAJ+19 Awad, M. A., Ashkiani, S., Johnson, R., Farach-Colton, M. and Owens, J. D., Engineering a high-performance gpu b-tree. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pages 145–157.

AFCO18 Ashkiani, S., Farach-Colton, M. and Owens, J. D., A dynamic hash table for the gpu. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pages 419–429.

AM10 Anh, V. N. and Moffat, A., Index compression using 64-bit words. *Software: Practice and Experience*, 40,2(2010), pages 131–147.

Bal Balevic, A., Fine-grain parallelization of entropy coding on GPGPUs, `http://tesla.rcub.bg.ac.rs/~taucet/docs/papers/HIPEAC-ShortPaper-AnaBalevic.pdf`. Retrieved 2020-02-09.

BH12 Bell, N. and Hoberock, J., Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*, Elsevier, 2012, pages 359–371.

BJS+02 Batzoglou, S., Jaffe, D. B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J. P. and Lander, E. S., Arachne: a whole-genome shotgun assembler. *Genome research*, 12,1(2002), pages 177–189.

DKGDG15 Deorowicz, S., Kokot, M., Grabowski, S. and Debudaj-Grabysz, A., KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31,10(2015), pages 1569–1576.

ERMH17 Erbert, M., Rechner, S. and Müller-Hannemann, M., Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms for Molecular Biology*, 12,1(2017), page 9.

KDD17 Kokot, M., Długosz, M. and Deorowicz, S., KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33,17(2017), pages 2759–2761.

KSS10 Kelley, D. R., Schatz, M. C. and Salzberg, S. L., Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11,11(2010), page R116.

MK11 Marçais, G. and Kingsford, C., A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27,6(2011), pages 764–770.

ND10 Nickolls, J. and Dally, W. J., The GPU computing era. *IEEE micro*, 30,2(2010), pages 56–69.

OHL+08 Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. and Phillips, J. C., GPU computing. *Proceedings of the IEEE*, volume 96, 2008, pages 879–899.

RLC13 Rizk, G., Lavenier, D. and Chikhi, R., DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29,5(2013), pages 652–653.

RRRS13 Raskhodnikova, S., Ron, D., Rubinfeld, R. and Smith, A., Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65,3(2013), pages 685–709.

SKIA14 Suzuki, S., Kakuta, M., Ishida, T. and Akiyama, Y., Accelerating identification of frequent k-mers in DNA sequences with GPU, `http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4190_bioinformatics_genome_assembly_GPU.pdf`, 2014. Retrieved 2020-04-26.

SPHC16 Shareghi, E., Petri, M., Haffari, G. and Cohn, T., Fast, small and exact: Infinite-order language modelling with compressed suffix trees. *Transactions of the Association for Computational Linguistics*, 4, pages 477–490.

SWYZ02 Scholer, F., Williams, H. E., Yiannis, J. and Zobel, J., Compression of inverted indexes for fast query evaluation. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, 2002, pages 222–229.

Appendix 1. Data sets

M. balbisiana

ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA098/SRA098922/SRX339427/SRR956987.fastq.bz2

ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA098/SRA098922/SRX339427/SRR957627.fastq.bz2

H. Sapiens

https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12878D_HiSeqX_R1.fastq.gz

https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12878D_HiSeqX_R2.fastq.gz

GRCh38

https://hgdownload.cse.ucsc.edu/goldenpath/hg38/bigZips/hg38.fa.gz